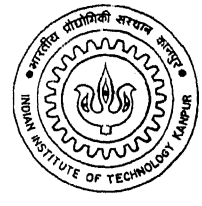
An Improvement on Distributed Resampling Routine used in IRS Data Processing Software

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

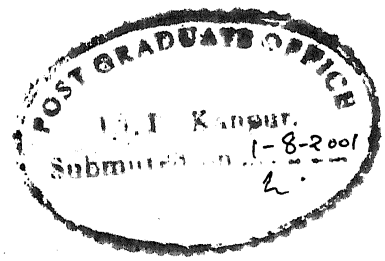
Onong Tayeng



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

August, 2001



Certificate

This is to certify that the work contained in the thesis entitled "*An Improvement on Distributed Resampling Routine used in IRS Data Processing Software*", by *Onong Tayeng*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

st, 2001

(Dr. Phalguni Gupta)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Serial algorithms for processing remotely sensed images generally take a prohibitive amount of time due to the large size of data. Parallelization of the algorithms and use of distributed computing methods is a possible means to improve the performance of the algorithms. Software packages like PVM and MPI are available which provide a platform for writing distributed applications. Many factors influence the performance of a distributed application. LAN bandwidth is one of the important factors. If the distributed application involves a large number of data transfers, then bandwidth becomes the bottleneck.

In this work, we have investigated the possibilities of using concepts from data compression to reduce the data size, thereby, reducing the data transfer time. We have implemented our ideas in the distributed resampling routine used by Space Applications Centre, Ahmedabad and obtained substantial performance improvement.

Acknowledgements

I would like to thank very much my guide Dr. Phalguni Gupta for his constant support and guidance throughout this work. I am very grateful to him for the insightful discussions and the patience he has shown throughout the work.

I would also like to thank my batchmates for making my stay memorable. Special thanks goes to the Mini-MTech99 batch of CC223 for the camaraderie, encouragement and the constant supply of tea, coffee and biscuits among other things.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Remote sensing, in the simplest words, means obtaining information about an object without touching the object itself. It has two facets: the technology of acquiring the data through a device which is located at a distance from the object, and analysis of the data for interpreting the physical attributes of the object, both these aspects being intimately linked with each other [16]. The data acquisition is done by sensors which can be photographic cameras, scanners or radiometers. The sensors are mounted on suitable platforms which could be of varying types: *terrestrial* (e.g. hydraulic platforms or as in hand-held field instruments), *aerial* (balloons, helicopters and aircrafts) or *space-borne* (rockets, manned and unmanned satellites). With the advancement in space-technology, satellites have become the major means of data gathering for remote sensing applications. Now onwards we will use the term remote sensing to stand for satellite remote sensing.

The basic principle involved in remote sensing methods is that in different wavelength ranges of the electromagnetic spectrum, each type of object reflects or emits a certain intensity of light, which is dependent upon the physical or compositional attributes of the object. Thus, using information from one or more wavelength ranges, it may be possible to differentiate between different types of objects(e.g. dry soil, vegetation, limonitic area etc.) and map their distribution on the ground [16].

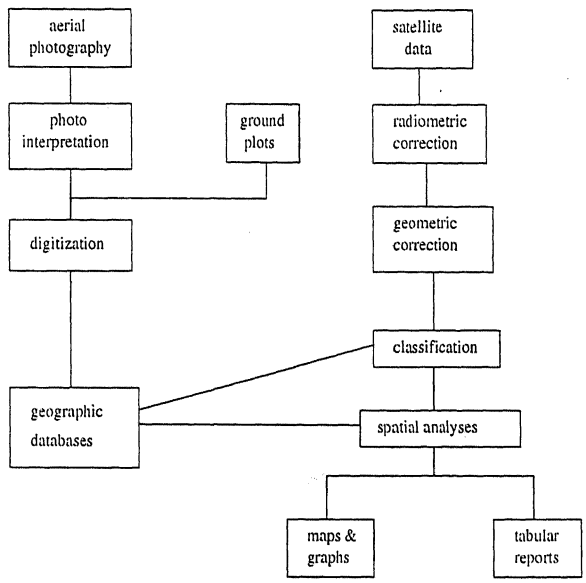


Figure 1.1: Remote sensing process

The sensors record the radiation intensity values for some range in the electromagnetic spectrum. For example, IRS LISS-I and LISS-II provide data in four spectral bands, namely 0.45-0.52μ, 0.52-0.59μ, 0.62-0.68μ and 0.77-0.86μ, while IRS LISS-III provide data in three main spectral bands, namely 0.52-0.59μ, 0.62-0.68μ and 0.77-0.86μ, and one optional band, 1.55-1.70μ. The data recorded by the sensors can be visualized as a two-dimensional array of spatial variations in brightness over an area or a two-dimensional image of the area. Each pixel value represents the average radiation intensity measured electronically over the ground area corresponding to the pixel. This intensity is represented in digital numbers(DNs). Digital numbers are positive integers(usually 0-255) resulting from the analog-to-digital conversion of the electrical signal measured by the sensors. The image data is transmitted to a receiving ground-station. Various corrections have to be performed on the image before proceeding to interpretations and applications. The choice of corrections depends on the requirements of the application and also on the sensor characteristics. Figure 1.1 shows the general remote sensing process.

The corrections required for remotely sensed images can be broadly categorized into: *Radiometric* and *Geometric*. Radiometric corrections refer to correcting errors that arise due to:

- ground/terrain properties,
- environmental factors during data acquisition and,
- sensor instrumentation factors

Geometric corrections are for correcting

- distortions related to platform altitude and perturbations,
- distortions related to sensor systems and,
- distortions related to earth's shape and spin.

The corrections are an essential part of the whole remote sensing process. The remotely sensed images would not be of much use without the corrections. Various algorithms have been proposed and implemented to perform the radiometric and geometric corrections. But due to the large size of data, they usually take a prohibitive amount of time. Parallelization of the algorithms and use of distributed computing methods is a possible means to achieving better performance. In any organization, there is usually a large number of small work-stations available connected through the LAN, creating an ideal environment for parallelisation and distributed computing. Software packages like PVM and MPI are available for developing distributed applications using computers connected through a network [15]. Distributed computing using PVM and MPI is used in solving important scientific, industrial and medical problems. A popular example of use of distributed computing in solving scientific problems is the SETI@Home project which uses the millions of computers on the internet to analyze the data for them while the user's computer is idle.

1.1 Goal of the Thesis

Resampling is the process of calculating new DN values for pixels created during geometric correction of remotely sensed images. The input to the resampling process is the radiometrically corrected image data. It is a very frequently used routine in space research organisations. Hence, it is desirable that it be efficient and fast. Distributed version of the resampling routine shows better performance than the serial routine by as much as 30-40% depending on the number of machines used. The distributed resampling model consists on N machines connected through the LAN. The collection of machines could be heterogeneous. PVM is used as the interface for message passing. One machine or host is designated as the *master* host and the rest are *slave* hosts. The input image data is present on the master and it controls the whole resampling process. The data is distributed among the hosts in the PVM configuration for processing.

For our particular problem, the input to the resampling process is the radiometrically corrected IRS LISS-III scene data. The pixel values are of type *single-precision floating point* and lie in the range of [0.00,255.00].

The distributed resampling routine presents a challenge in terms of further improvement in the design as well as efficiency of the routine. We have observed that a substantial portion of the time is spent in I/O and in transferring data from the input data source to the other hosts apart from the time taken in processing the data. The input image size is usually very large, in the order of 350MB per scene. Data transfer, therefore, becomes a bottleneck. In this work, we have investigated the possibilitcies of using data compression techniques in the distributed model to reduce the data transfer overhead. Data compression refers to the art or science of representing information in compact form [3]. Basically, compression algorithms seek to reduce the number of bits used to store or transmit information. But compression has its costs in terms of time complexity. We also have to take into account the fact that the input image data is in the form of floating point numbers. Our objective is to design efficient algorithms which compress the input data with minimal time complexity.

Moreover, in a heterogeneous environment, there arises the issue of data incompatibility. Different computer architectures have different underlying data formats for representing multiple-byte data. Depending on the ordering of bytes in memory, we have *little-endian* and *big-endian* architectures [23]. In *little-endian* architecture, the leftmost bytes (those with lower address) are most significant, while in *big-endian* architectures, the rightmost bytes are most significant. For example, consider the binary representation of the number 1025 (assuming that 4 bytes are assigned for an integer)

0000000 00000000 00000100 00000001

and the way it is stored in the big-endian and little-endian formats,

Address	Big-Endian	Little-Endian
00	00000000	00000001
01	00000000	00000100
02	00000100	00000000
03	00000001	00000000

Thus, in a heterogeneous environment, the same sequence of bytes could acquire different values on different architectures. This can to be avoided by encoding the message in an architecture independent format. PVM uses the XDR [18] encoding when messages have to be exchanged in a heterogeneous environment. We will show that our algorithm eliminates the need for the XDR encoding done by PVM thus eliminating that overhead.

1.2 Organization of thesis

Following is the layout of this report:

Chapter 2. This chapter describes the *resampling* process and the distributed resampling routine.

Chapter 3. In this chapter, we have tried to give a brief overview of the topic of data compression.

Chapter 4. The IEEE 754 standard for floating point numbers is described in brief and a survey of work done in the area of floating point data is provided.

Chapter 5. In this chapter, we present the approaches we have used and results from tests.

Chapter 6. Conclusions and future work scope is presented.

Chapter 2

Distributed Resampling Model

2.1 Introduction

One of the corrections that is done on the image data received from remote sensing satellites is *geometric correction.* It transforms the geometry of the image and is concerned with the relationship between the coordinates of the object and its pixel location in the image [8]. It is necessary to relate the image to the other geocoded data such as maps. Geometric errors fall into three main categories:

- Error due to Instability of the Platform - It is due to the variation in the altitude of the satellite.
- Error due to instrument - distortions in optical system, nonlinearity of the scanning mechanism and non-uniform sampling rates.
- Error due to Earth's rotation and shape.

We are concerned with the error caused due to the Earth's rotation in this work. The image scanned is rectangular but the area that it represents is of the shape of a parallelogram as shown in Figure 2.1. So, the co-ordinate system of the image (x-y) has to be geometrically transformed to a map projection co-ordinate system (u-v). As input, we have the pixel valued in the lattice of x-y co-ordinate system and as

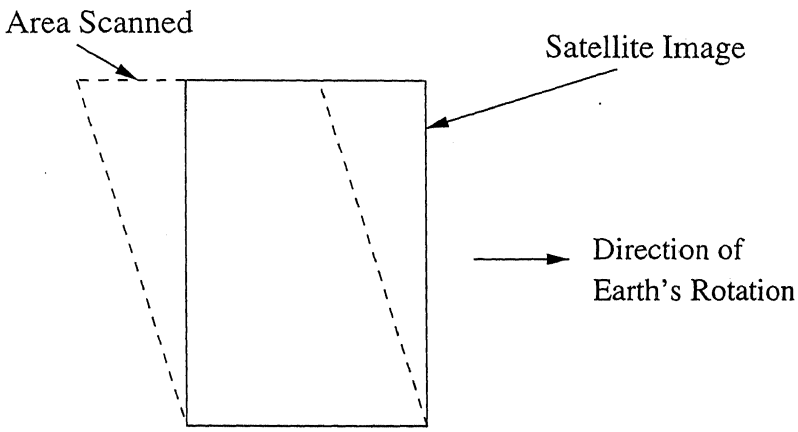


Figure 2.1: Skew caused by Earth's rotation.

output, we have to generate the corresponding pixel values at the lattice points in the u-v coordinate system. The problem is called *Resampling.*

Resampling requires a transformation that maps the pixels in the u-v co-ordinate system to points in the x-y coordinate system as shown in Figure 2.2. The plus (+) and circle (◯) symbols denote the grid points in the x-y and u-v co-ordinate system respectively. The triangles (△) denote the position of the grid points of the u-v co-ordinate system in the x-y co-ordinate system. The transformation is found out using the ground control points. A ground control point is a physical feature detectable in the image and whose precise location is known [9]. Differences between the actual and the observed ground control point locations can be used to model the geometric error and hence the transformation can be found out.

Once the points from the u-v co-ordinate system are mapped onto the x-y co-ordinate system, we evaluate the intensity at the grid point in the u-v co-ordinate system by interpolation. The simplest and the easiest interpolation function is the *Nearest Neighborhood* approach. The intensity value at the grid point in the u-v co-ordinate system is the intensity value of the pixel in the x-y co-ordinate system that is nearest to its corresponding mapped point in the x-y co-ordinate system [9].

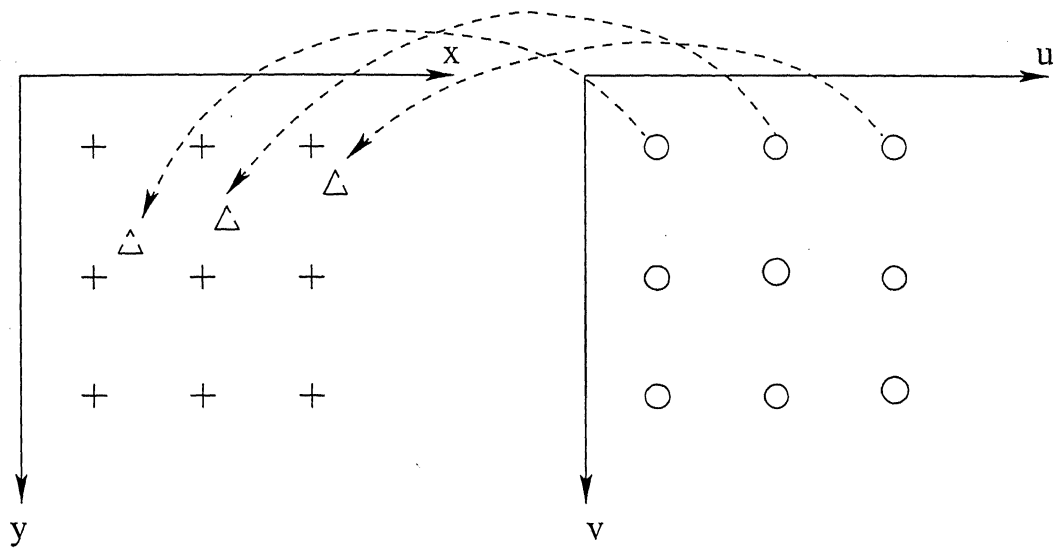


Figure 2.2: Transformation from x-y to u-v co-ordinate system

This method is very efficient to compute but has the drawback of producing blocky effect. A better approach is to consider the entire process in the Fourier transform [1] domain. Resampling can be written as a convolution

$$g(x) = \int f_s(x)s(x-r)dr \quad ,$$

where $f_s(x)$ is the original sampled function, $s(x)$ is the interpolation function and $g(x)$ is the attempted recovery of the original continuous image. In Fourier transform domain, convolution becomes multiplication and hence

$$G(w) = F_s(w)S(w) \quad ,$$

where $F_s(w)$ is the Fourier transform of the original image, $S(w)$ is the Fourier transform of the interpolation function and $G(w)$ is the Fourier transform of the recovered image. The ideal interpolation function for this purpose is the *sinc* function, $\sin(x)/x$. The algortihm which uses a cubic polynomial approximation of the *sinc* function is called the *Cubic Convolution* algorithm.

2.2 Implementation Environment

The distributed resampling routine has been developed with PVM as the underlying interface for message passing and process control. The routine has been tested at Space Applications Centre, Ahmedabad as well as in our departmental lab. At our lab we have four 256MB PentiumIII PCs connected through Intel 10/100Mbps switch. At Space Applications Centre, Ahmedabad, we had two 256MB PentiumIII's and one 64MB PentiumII machine connected over 100Mbps LAN. At both places PVM version 3.4.2 was used and relevant information about PVM is provided in Appendix A.

2.3 PVM Configuration

Before the application is run on the given environment, it has to be first configured. Configuring the PVM means adding machines, also called hosts, to the PVM so that sitting on a single machine, we are given the feel of controlling a parallel machine over the network. We can change the configuration of PVM, if desired, by adding or deleting hosts from the environment before each execution of the resampling routine.

In a PVM configuration, each process has a unique *task id* (or tid) over the network. This tid is used to address the process while sending or receiving the data. So, every process can communicate with every other process on PVM provided it has the tid of the process with which it wishes to communicate. Also, a process that spawns another process is called the parent of the spawned process.

2.4 Description of the Model

The model is a typical *master-slave* model. Initially we configure PVM environment, adding the hosts over the network. The master process reads in the PVM configuration and spawns processes on the hosts. The machine on which the master process runs is called the *master* host. All other machines in the PVM configuration are the *slave* hosts. This is shown in Figure 2.3.

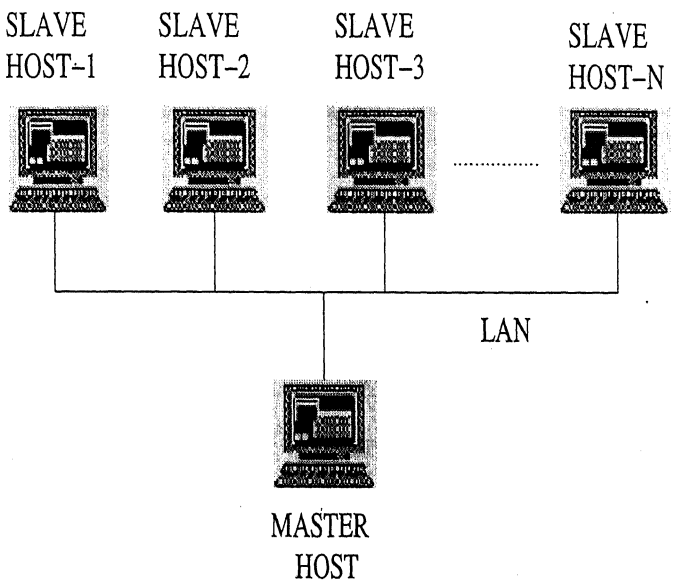


Figure 2.3: Master-Slave machines over the LAN

The input LISS III 4-band data files reside on the master host and the resampled output data generated is stored on a separate host which could be any one of the slave hosts and is called the *Receiver*. The processing is done in bunches. A bunch of data is read at the master and sent to one of the hosts where processing is done. The resampled output is then sent to the Receiver where it is written back to the disk. The processing part is done in parallel on the various slave hosts, i.e., the master sends data to several slaves which process it in parallel and the Receiver receives the processed data from the slaves.

There are a number of routines in the parallel distributed resampling module. There is a master process running on the master host which controls the whole operation. A number of *Raddata* processes run on the master host. Each Raddata process reads a bunch of radiometrically corrected data kept on the master host. Also, a number of *ResampProc* processes run on various slave hosts as well as the master host. These ResampProc processes take data from the corresponding Raddata processes, process the data and return the processed data to the *Receiver*.

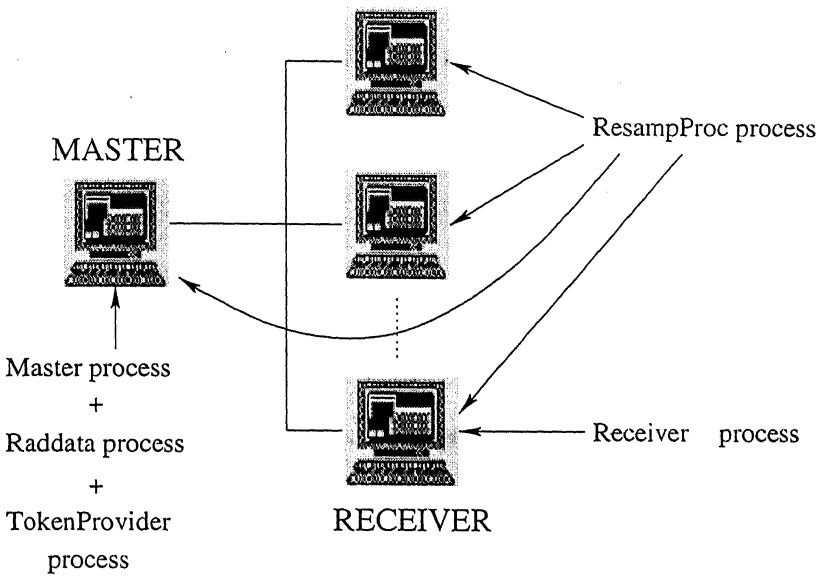


Figure 2.4: Process distribution over the PVM configuration.

Apart from these processes, there is a *TokenProvider* process on the master host that serializes the reads of the raddata process. The Figure 2.4 shows the distribution of the processes on the hosts configured in the PVM environment.

2.5 Description of Routines

2.5.1 Master Process

Once the PVM environment is configured, the master process is executed on a host. This host will be referred to as the master host. The master process reads the PVM configuration and spawns the following processes :

- Receiver process on one of the slave hosts
- TokenProvider process on the master host

The master process computes some lookup tables required in the resampling process and performs other initialization tasks. Then for each host in the PVM configuration, it spawns a Raddata process on the master host and a corresponding

ResampProc process.

Apart from spawning processes, the master process also sends relevant data to the needy processes. For example,

- sends tids of Raddata processes to the TokenProvider process and vice-versa
- sends look-up tables and other necessary data to ResampProc processes
- sends tid of the Receiver Process to the ResampProc processes and vice-versa

These data helps the processes in doing their job and communicating with other processes. In short, the master process acts as a kind of monitor of the whole parallel execution system.

2.5.2 Raddata Process

The Raddata process reads data from the input files as instructed by the master process. Then it sends the data to its corresponding ResampProc process on one of the hosts. Since the master spawns many Raddata processes simultaneously, they may compete against each other for reading the data from the hard disk. To prevent this, the Raddatas are serialized by allowing only one active Raddata process at a time in the master host. This is achieved with the help of TokenProvider process. The Raddata processes wait for a token from the TokenProvider process and only the process that has the token can read its data from the disk. The Raddata processes sleep till they receive token from the TokenProvider process.

2.5.3 Tokenprovider Process

The master process passes the tids of all the Raddata processes on the master host, to this process. It gives a token to the first Raddata process. When that Raddata process finishes reading its data, it sends the token back to the TokenProvider process. The TokenProvider process then passes the token to the next Raddata process. This continues till all Raddatas have completed reading their data. Using this approach, the disk reads on the master host are serialized.

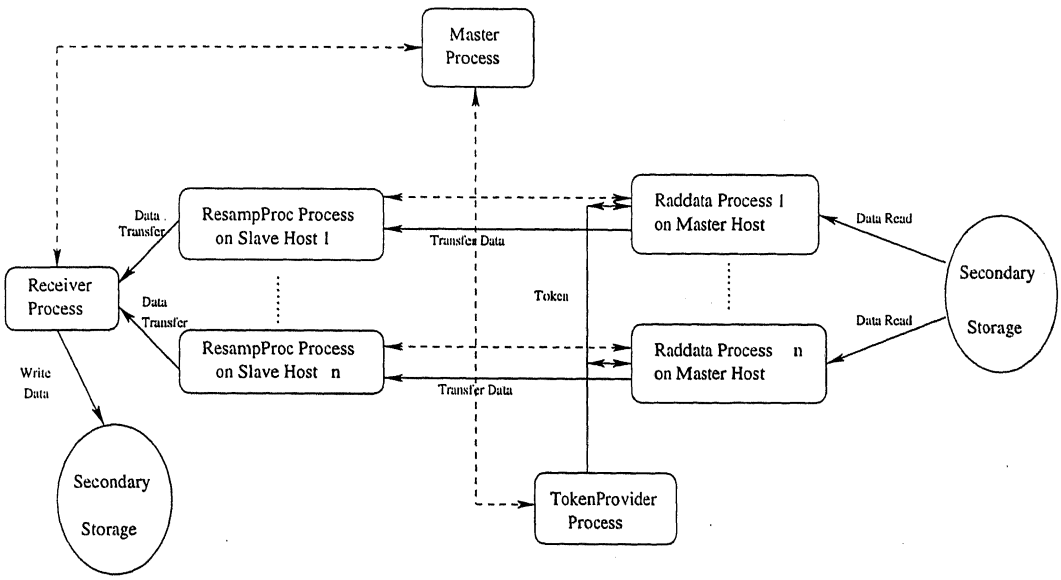


Figure 2.5: Resampling Model.

2.5.4 ResampProc Process

The actual processing of the data is done by the ResampProc processes. The ResampProc process gets its data from the corresponding Raddata process. It performs resampling of the data and the processed data is sent to the Receiver process.

2.5.5 Receiver Process

The *Receiver* process can be spawned on any one of the slave hosts. The job of the *Receiver* is to receive the resampled data from the *ResampProc* processes and write them onto the disk. After it has received the output data corresponding to the bunches, it sends a special tag to the *Master* indicating the same and exits.

Figure 2.5 shows various processes and their interaction over the LAN when we have a master host and N slave hosts.

2.6 Experimental Results

The performance of any routine over PVM is not guaranteed. At best, one can present the performance results under ideal conditions. Here, ideal conditions mean that : (1) network load is minimum, i.e., there is no network congestion at any stage during the execution of the routine and very few users are transferring data over the network during the period, (2) machine load is minimum, i.e., there is no user working on the hosts that are configured into the PVM environment and, (3) there is no host or task failure during the execution of the routine.

We tested the distributed resampling module under the above ideal conditions with four different configurations at our labs. The details of these configurations are given below :

- Configuration 1 : A single PentiumIII PC with 256MB RAM.
- Configuration 1 : Two PentiumIII PCs with 256MB RAM, connected through 100Mbps switch.
- Configuration 1 : Three PentiumIII PCs with 256MB RAM, connected through 100Mbps switch.
- Configuration 1 : Four PentiumIII PCs with 256MB RAM, connected through 100Mbps switch.

The table below gives the performance of the routine under different configurations :

Configurations	Average execution time
Configuration 1	4 minutes 45 seconds
Configuration 2	3 minutes 30 seconds
Configuration 3	3 minutes 1 seconds
Configuration 4	2 minutes 40 seconds

Table 2.1: Performance comparison under different configurations

2.7 Summary

It is clear from the description of the resampling routine that a lot of data has to be sent and received over the network. The LAN bandwidth affects the performance of the routine. A faster LAN will improve the performance. Faster computers and the amount of memory also play a very important role in the performance of the routine.

Chapter 3

Brief Introduction to Data Compression

Information management is becoming increasingly important as the quantity of information is growing at an exponential rate. By information here we mean information in electronic form, i.e., in the form of *bits* or *bytes*. These could be anything from the records in a computer database, executable program files, text files, images, output from some experiment etc. Huge amount of information have to be stored. Another problem that arises is that of data transfer. When data has to be transmitted through a channel, the size of the data affects the amount of time required. This has led to the development of a variety of algorithms which *compress* data. Real world data files generally have redundancies. Text files have redundancies in the form of repeating words or letters, in images pixel values tend to be co-related and so on. Compression algorithms try to take advantage of the redundancies to reduce the number of bits required to store the data.

In this chapter, we'll briefly describe the important aspects of data compression.

3.1 Introduction

A compression algorithm which only causes compression would not be of much use. There has to be some way of recovering the original data from the compressed data.

Consequently, a compression algorithm consists of two complementary algorithms, one which encodes the data into compact form and the other, which generates the original from the compressed data. Figure 3.1 illustrates the compression process.

Based on the requirements of reconstruction,data compression schemes can be divided into two broad classes: *lossless* compression schemes and *lossy* compression schemes.

3.1.1 Lossless Compression

Lossless compression techniques involve no loss of information.If data has been losslessly compressed, the original data can be recovered exactly from the compressed data. Lossless compression is generally used for applications that cannot tolerate any difference between the original and reconstructed data.

Text compression is an important area for lossless compression. It is very important that the reconstruction is identical to the original text, as very small differences can result in statements with very different meanings. Consider the sentences "Do not send money" and "Do now send money." A similar argument holds for computer files and for certain types of data such as bank records. Examples are Huffman coding, Arithmetic coding, LZ77 and LZ78 algorithms.

3.1.2 Lossy Compression

Lossy compression techniques involve some loss of information, and data that have been compressed using lossy techniques generally cannot be recovered or reconsructed exactly. In return for accepting this distortion in reconstruction, we can generally obtain much higher compression ratios than is possible with lossless compression. An example is the well known JPEG.

3.1.3 Measures of Performance

A compression algorithm can be evaluated in a number of ways. We could measure the relative complexity of the algorithm, the memory requirement for implementing

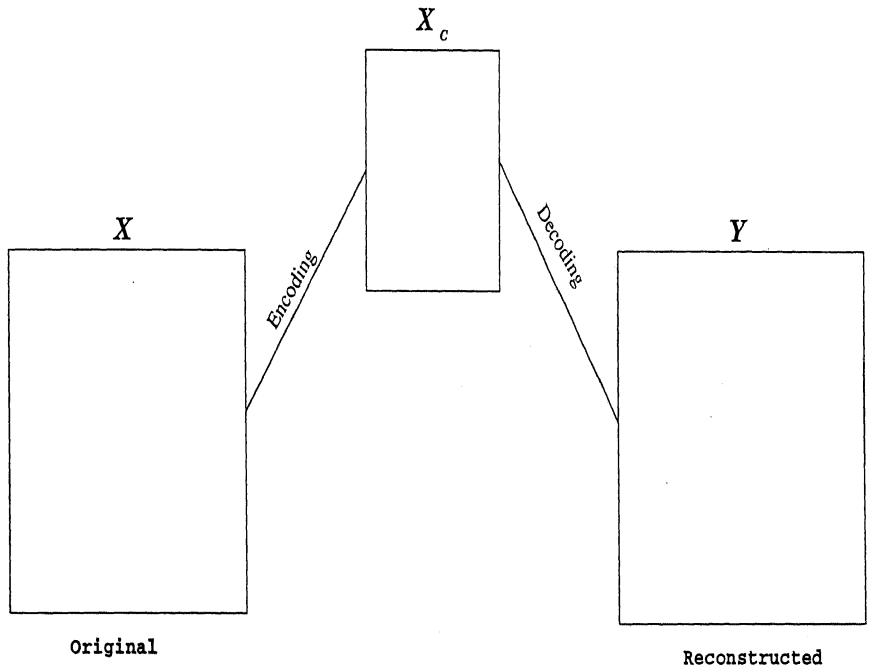


Figure 3.1: Compression and Decompression

the algorithm, speed of execution, the amount of compression, and how closely the reconstruction resembles the original.

Compression ratio is the ratio of the number of bits required to represent the data before compression to the number of bits required to represent the data after compression.

$$Compression\ Ratio = \frac{No.\ of\ bits\ before\ compression}{No.\ of\ bits\ required\ after\ compression}$$

A 256 x 256 image requires (assuming 1 byte per pixel) would require 65,356 bytes. Suppose that after compressed representation requires 16,384 bytes, then the compression ratio is 4:1. We can also represent the compression ratio by expressing the reduction in the amount of data required as a percentage of the size of the original data. So, in this particular example the compression ratio calculated in this manner would be 75%.

3.2 Information Theory

Information theory deals with the quantitative measure of information. It involves the study of efficient encoding of information and its consequences in the form of speed of transmission and probability of error. It provides a means of estimating the amount of compression that can be achieved given the statistical nature of the data.

3.2.1 Self-Information and Entropy

Claude Elwood Shannon [2], an electrical engineer at Bell Labs, is credited with the present form of information theory. Shannon defined a quantity called *self-information.* Suppose we have an event A, which is a set of outcomes of some random experiment. If $P(A)$ is the probability that the event A will occur, then the self-information associated with A is given by

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A) \log_b(1) = 0 \tag{1}$$

The value of -log(x) increases as x decreases from one to zero. Therefore, if the probability of an event is high, the information associated with it is low; if the probability of an event is low, the information associated with it is high. The unit of information depends on the base of the log.If we use log base 2, the unit is *bits*; if we use log base e, the unit is *nats*; and if we use log base 10, the unit is *hartleys*. A real-world example would be a comparison between the two statements:

1. it is raining.

2. the moon of earth has exploded.

The first case happens every once in a while (assuming we are not living in a desert area). Its probability may change around the world, but may be something like 0.3 during bleak autumn days. One would not be very surprised to hear that it is raining outside. In other words, the first statement doesn't contain much information. It is not so for the second case. The second case would be big news,

as it has never before happened, as far as we know. Although it seems very unlikely we could decide a very small probability for it, like 10^{-30}. Taking log base 2, the equation gives the self-information for the first case as 1.74 bits, and 99.7 bits for the second case.

If we have a set of independent events A_i, which are sets of outcomes of some experiment $\mathcal{S}$, such that

$$\bigcup A_i = S$$

where S is the sample space, then the average self-information associated with the random experiment is given by

$$H = \sum P(A_i) i(A_i) = -\sum P(A_i) \log_b P(A_i) \tag{2}$$

This quantity is called the *entropy* associated with the experiment. Shannon showed that if the experiment is a source that puts out symbols A_i from a set $\mathcal{A}$, then the entropy is a measure of the average number of bits needed to code the output of the source. He showed that the best that a lossless compression scheme can do is to encode the output of a source with an average number of bits equal to the entropy of the source.

3.2.2 Symbol Sources

Given a source, the set of symbols $\mathcal{A}$ is called the *alphabet* for the source, and the symbols are referred to as *letters*. Information theory traditionally deals with symbol sources that have certain properties. One important property is that they give out symbols that belong to a finite, predefined alphabet $\mathcal{A}$. An alpbhabet could consist of all the letters of the english alphabet, numeric values in some particular range etc.

- $\mathcal{A}$ = a set of English letters = {a,b,c.....,z,A,B.....Z}
- $\mathcal{A}$ = set of binary digits = {0,1}
- $\mathcal{A}$ = set of integers in the range [0,N] = {0,1,2,.....,N}

In the real world, symbol sources are computer files which contain data of various kinds. The basic unit of information is the *byte* whose value lies in the range [0,255]. Hence, the symbol source can be thought to be containing integers in the range [0,255] or $\mathcal{A}$={0,1,..,255}.

3.2.3 Message Entropy

When reading symbols from a source, there is some probability associated with the occurence of each symbol. The entropy for a message sequence read from the source can be computed using (2).

Let us consider an example sequence: 1 2 3 2 3 4 5 4 5 6 7 8 9 8 9 10

Assuming that the frequency of occurence of each number is reflected accurately in the number of times it appears in the sequence, we can compute the probability of occurence of each symbol as follows:

$P(1) = P(6) = P(7) = P(10) = \frac{1}{16}$

$P(2) = P(3) = P(4) = P(5) = P(8) = P(9) = \frac{2}{16}$

Using (2), the entropy for this message sequence is 3.25 bits. This means that the best scheme we could find for coding this sequence could only code it at 3.25 bits/sample.

3.3 Coding

A code is any mapping from an input alphabet to an output alphabet. We will restrict ourselves with a binary output alphabet, i.e., the output alphabet will consist of binary digits or *bits*. Hence, by *coding*, we mean the assignment of binary sequences to elements of an alphabet. The individual members of the code are called *codewords*. An example would be the ASCII code. The ASCII codeword for the letter a is 1000011, while that of letter A is 10000001. Codes like ASCII codes which use the same number of bits to represent each symbol are called *fixed-length code*.

3.3.1 Average Codelength

Suppose we have a source alphabet $\mathcal{A} = \{a_1, a_2, ..., a_N\}$, the *average length* of a code for $\mathcal{A}$ is denoted by l is given by

$$l = \sum_{i=1}^{N} P(a_i)n(a_i) \tag{3}$$

where $n(a_i)$ is the number of bits in the codeword for symbol a_i, $P(a_i)$ is the probability of occurence of the symbol a_i and N is the size of the alphabet $\mathcal{A}$. The average length is given in bits/symbol and it denotes the average number of bits that would be required to encode the symbols in $\mathcal{A}$.

3.3.2 Uniquely Decodability

A useful coding scheme must be able to convey information in an unambiguous way. Suppose we have a source alphabet $\mathcal{A}$ consisting of four symbols a_1, a_2, a_3 and a_4, with probabilities $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, and $P(a_3)$=$P(a_4) = \frac{1}{8}$. The entropy for this source is 1.75 bits/symbol. Let us consider four different codes for this source

Symbols	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111

The average code lenght of each of the codes is given by

Code	l
Code 1	1.125
Code 2	1.25
Code 3	1.75
Code 4	1.875

Based on the average code length, Code 1 might appear to be the best code. But the code must be able to transfer information in an unambiguous manner. Code 1 does not satisfy this criteria bacause both a_1 and a_2 have been assigned the codeword 0. So, when a 0 is received, there is no way to know whether an a_1 was transmitted or an a_2.

Code 2 does not seem to have the problem of ambiguity. Each symbol is assigned a unique codeword. However, suppose, we want to encode the sequence $a_2a_1a_1$. Code 2 would encode it as the binary string 100. When this is received at the decoder , there's no way to know whether the sequence was $a_2a_1a_1$ or a_2a_3. It is desirable that a code is *uniquely decodable*, i.e., any given sequence of codewords can be decoded in one, and only one, way.

Code 3 and Code 4 are uniquely decodable. In Code 3, the first three codewords end in a 0 and the fourth codeword has three 1's. So, the decoder would go on accumulating bits until a zero is encountered or three 1's are encountered. In Code 4, each codeword starts with a 0. So, the decoder would know that it has decoded one symbol as soon as it encounters a 0.

3.3.3 Instantaneous Codes

A code is *instantaneous* if each codeword in a message can be decoded as soon as it is received. The binary code $\{a_1, a_2\} = \{0, 01\}$ is uniquely decodable, but it isn't instantaneous. We need to peek into the future to see if the next bit is 1. If it is, a_2 is decoded, otherwise, a_1 is decoded. The code $\{a_1, a_2, a_3\} = \{0, 10, 11\}$ on the other hand is an instantaneous code. Code 3 from above example is also instantaneous.

3.3.4 Prefix Codes

A code is a *prefix code* if and only if no codeword is a prefix of another codeword. A code is instantaneous if and only if it is a prefix code, so a prefix code is always a uniquely decodable instantaneous code. We will only deal with prefix codes from now on. It can be proven that all uniquely decodable codes can be changed into prefix codes of equal code lengths [3].

3.4 Code Classification

Codes can be divided into four groups:

1. Block-to-block codes
2. Block-to-variable codes
3. Variable-to-block codes
4. Variable-to-variable codes

3.4.1 Block-to-block codes

These codes take a specific number of bits at a time from the input and emit a specific number of bits as a result. If all of the symbols in the input alphabet (in the case of bytes, all values from 0 to 255) are used, the output alphabet must be the same size as the input alphabet, i.e. uses the same number of bits. Otherwise it could not represent all arbitrary messages. An example of such a code is the ASCII code.

3.4.2 Block-to-variable codes

Block-to-variable codes use a variable number of output bits for each input symbol. All statistical data compression systems, such as symbol ranking, Huffman coding, Shannon-Fano coding, and arithmetic coding belong to this group. The idea is to assign shorter codes for symbols that occur often, and longer codes for symbols

that occur rarely. This provides a reduction in the average code length, and thus compression.

3.4.3 Variable-to-block codes

The previous compression methods handled a specific number of bits at a time. A group of bits are read from the input stream and some bits are written to the output. Variable-to-block codes behave just the opposite. They use a fixed-length output code to represent a variable-length part of the input. Variable-to-block codes are also called *free-parse* methods, because there is no pre-defined way to divide the input message into encodable parts (i.e. strings that will be replaced by shorter codes). Substitutional compressors belong to this group. Substitutional compressors work by trying to replace strings in the input data with shorter codes. The Lempel-Ziv family of algorithms belong to this category [13][14].

3.4.4 Variable-to-variable codes

The compression algorithms in this category are mostly hybrids or concatenations of the previously described compressors. For example a variable-to-block code such as LZ77 followed by a statistical compressor like Huffman encoding falls into this category and is used in Zip, LHa, Gzip and many more. They use fixed, static, and adaptive statistical compression, depending on the program and the compression level selected.

3.5 Lossless Examples: RLE and Huffman Coding

3.5.1 Run-Length Encoding

Run-Length encoding or *RLE* is a very simple to implement and fast compression algorithm though its performance is not as good as other algorithms. It is useful when there are long runs of the same symbol in the input.

If there are consecutive equal valued symbols in the input, the encoder simply outputs how many of them there are, and their value. But, there must be some

way to identify literal bytes and compressed data. Some of the RLE compressors output two equal symbols to identify a run of symbols. The next byte(s) then tell how many more of these to output. If the value is 0, there are only two consecutive equal symbols in the original stream. Depending on how many bits are used to represent the value, this is the only case when the output is expanded. Other RLE compressors use a special control symbol to indicate a run of symbols, followed by the symbol itself and lastly the length of the run. This method achieves compression only if there are runs of length 4 or more.

Suppose the following string of data (17 bytes) has to be compressed:

ABBBBBBBBBCDEEEEF

Using RLE compression, the compressed file takes up 10 bytes and could look like this:

A*9BCD*4EF

As we can see, repetitive strings of data are replaced by a control character (*) followed by the number of repeated characters and the repetitive character itself. The control character is not fixed, it can differ from implementation to implementation.

RLE is fast because it deals with byte-aligned data and is essentially just copying bytes from one place to another. The drawback is that RLE can only compress identical bytes into a shorter representation.

3.5.2 Huffman Coding

Let us consider a language with a simple alphabet, $\mathcal{A} = \{ a, b, c, d, e \}$, and suppose that we have to encode the following message using the Huffman coding:

beeedacecabaeeee

A direct binary code would map the different symbols to consequtive bit patterns, such as:

Symbol	Code
a	000
b	001
c	010
d	011
e	100

Because there are five symbols, we need 3 bits to represent all of the possibilities, but we also don't use all the possibilities. Only 5 values are used out of the maximum 8 that can be represented in 3 bits. With this code the original message takes 48 bits:

$$beeedacecabaeeee = 001\ 100\ 100\ 100\ 011\ 000\ 010\ 001\ 010\ 000\ 001\ 000\ 100\ 100\ 100\ 100$$

For Huffman coding the message and for entropy calculation, we first need to calculate the symbol frequencies from the message. The probability for each symbol is the frequency divided by the message length. When we reduce the number of bits needed to represent the probable symbols (their code lengths) we can also reduce the average code length and thus the number of bits we need to send.

Symbol	Frequency/Total	Probability
d	$\frac{1}{16}$	0.0625
a	$\frac{3}{16}$	0.1875
e	$\frac{8}{16}$	0.5
b	$\frac{2}{16}$	0.125
c	$\frac{2}{16}$	0.125

The entropy gives the lower limit for statistical compression method's average codelength. Using the equation 2 from the next section, we can calculate it as 1.953. This means that however cleverly we select a code to represent the symbols, in average we need at least 1.953 bits per symbol.

Next we create the Huffman tree. We first rank the symbols in decreasing probability order and then at each step combine two lowest-probability symbols into a

single composite symbol (C1, C2, ..). The probability of this new symbol is therefore the sum of the two original probabilities. The process is then repeated until a single composite symbol remains:

Step 1	Step 2	Step 3	Step 4
e 0.5	e 0.5	e 0.5	**C3** 0.5→**C4**
a 0.1875	**C1** 0.1875	**C2** 0.3125→**C3**	e 0.5
b 0.125	a 0.1875→**C2**	**C1** 0.1825	
c 0.125→ **C1**	b 0.125		
d 0.0625			

At each step two lowest-probability nodes are combined until we have only one symbol left. The corresponding huffman tree is shown in Figure 3.2. We start at the final symbol (C4 in this case), break up the composite symbol assigning 0 to the first symbol and 1 to the second one. The huffman tree eliminates the need to store the symbol probabilities.

Symbol	Codeword	Codeword Length
a	000	3
b	001	3
e	1	1
c	010	3
d	011	3

When we follow the tree from to top to the symbol we want to encode and remember each decision (which branch to follow), we get the code: 'C', 'D', 'I', 'S', 'V' = 011, 000, 1, 001, 010. For example when we see the symbol 'C' in the input, we output 011. If we see 'I' in the input, we output a single 1. The code for 'I' is very short because it occurs very often in the input.

Now we have the code lengths and can calculate the average code length: $0.0625 \times 3 + 0.1875 \times 3 + 0.5 \times 1 + 0.125 \times 3 + 0.125 \times 3 = 2$. We did not quite reach the lower limit that entropy gave us. Well, actually it is not so surprising because we know that Huffman code is optimal only if all the probabilities are negative powers

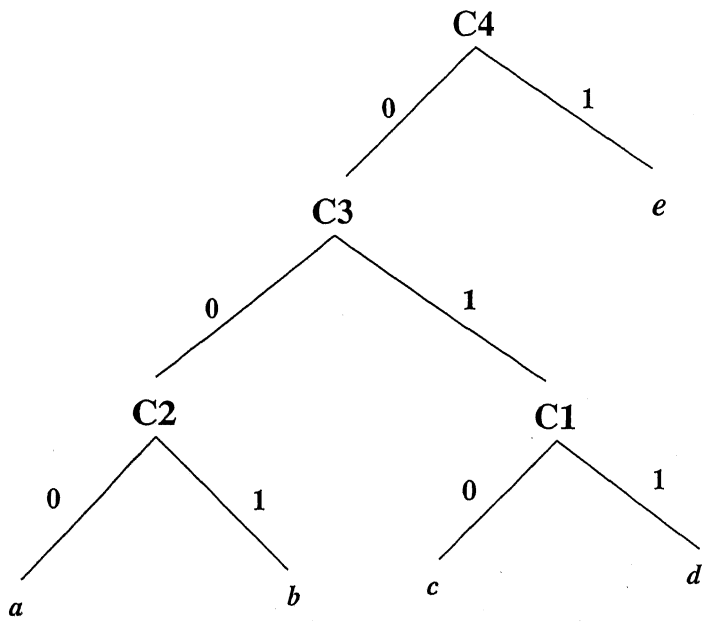


Figure 3.2: Building the Huffman Tree

of two.

Encoded, the message becomes:

$$beeedacecabaeeee = 001\ 1\ 1\ 1\ 011\ 000\ 010\ 1\ 010\ 000\ 001\ 000\ 1\ 1\ 1\ 1$$

So, the compressed output takes 32 bits. The message originally took 48 bits.

3.6 Lossy Compression Example : JPEG

Lossy compression is fundamentally diferent from lossless compression in one respect: it accepts a slight loss of data to facilitate compression. Lossy compression is generally done on analog data stored digitally, with the primary application being graphics and sound files. Lossy algorithms generally make two passes over the input data. A first pass over the data performs a high-level, signal-processing function. This frequently consists of transforming the data into frequency domain, using

algorithms similar to Fast Fourier Transform(FFT). Once the data has been transformed, it is smoothed or quantized, rounding off high and low points. Loss of signal occurs here. Finally, the quantized values are compressed using conventional lossless techniques. A very brief description of the JPEG lossy algorithm is given as an example.

The JPEG lossy algorithm operates in three successive stages, shown in Figure 3.6.

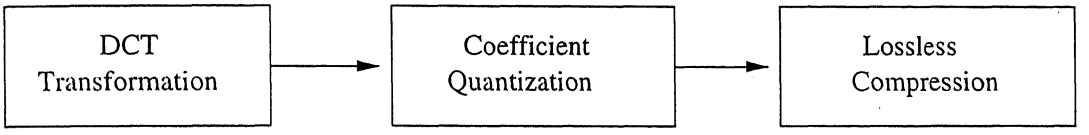


Figure 3.3: JPEG Lossy Compression

In the first stage, the Discreet Cosine Transformation (DCT) is applied to the input and the DCT coefficients are obtained. The inverse DCT gives back the original pixel values. The goal of the DCT is to transform the image into the frequency domain. In the second stage, the DCT coefficients are quantized. This leads to loss of information but allows high degree of compression to be achieved. The quantization constant can be user specified. Lastly, the quantized values are further compressed by using a lossless compression scheme like huffman coding or RLE.

3.7 Representing Integers

Many compression algorithms use integer values for something or another. Any algorithm that needs to represent integer values can benefit very much if we manage to reduce the number of bits needed to do that. This is why efficient coding of these integers is very important. What encoding method to select depends on the distribution and the range of the values.

3.7.1 Fixed, Linear

If the values are evenly distributed throughout the whole range, a direct binary representation is the optimal choice. The number of bits needed depends on the range. If the range is not a power of two, some tweaking can be done to the code to get nearer the theoretical optimum $\log_2(range)$ bits per value.

Value	Binary	Adjusted 1	Adjusted 2
0	000	00	000
1	001	01	001
2	010	100	010
3	011	101	011
4	100	110	10
5	101	111	11

The above table shows two different versions of how the adjustment could be done for a code that has to represent 6 different values with the minimum average number of bits. As can be seen, they are still both prefix codes, i.e. it's possible to (easily) decode them.

If there is no definite upper limit to the integer value, direct binary code can't be used and one of the following codes must be selected.

3.7.2 Elias Gamma Code

The Elias gamma code assumes that smaller integer values are more probable. In fact it assumes (or benefits from) a proportionally decreasing distribution. Values that use n bits should be twice as probable as values that use $n+1$ bits.

In this code the number of zero-bits before the first one-bit (a unary code) defines how many more bits to get. The code may be considered a special fixed Huffman tree. We can generate a Huffman tree from the assumed value distribution and get a very similar code. The code is also directly decodable without any tables or difficult operations, because once the first one-bit is found, the length of the code word is

instantly known. The bits following the zero bits (if any) are directly the encoded value.

Gamma Codeword	Integer	Bits
1	1	1
01x	2-3	3
001xx	4-7	5
0001xxx	8-15	7
00001xxxx	16-31	9
000001xxxxx	32-63	11
0000001xxxxxx	64-127	13

3.7.3 Elias Delta Code

The Elias Delta Code is an extension of the gamma code. This code assumes a little more 'traditional' value distribution. The first part of the code is a gamma code, which tells how many more bits to get (one less than the gamma code value).

Delta Code	Integer	Bits
1	1	1
010x	2-3	4
011xx	4-7	5
00100xxx	8-15	8
00101xxxx	16-31	9
00110xxxxx	32-63	10
00111xxxxxx	64-127	11

The delta code is better than gamma code for big values, as it is asymptotically optimal (the expected codeword length approaches constant times entropy when entropy approaches infinity), which the gamma code is not. What this means is that the extra bits needed to indicate where the code ends become smaller and smaller proportion of the total bits as we encode bigger and bigger numbers. The

gamma code is better for greatly skewed value distributions (a lot of small values).

3.7.4 Golomb Codes

Golomb codes are prefix codes that are suboptimal (compared to Huffman), but very easy to implement [20]. Golomb codes are distinguished from each other by a single parameter m. This makes it very easy to adjust the code dynamically to adapt to changes in the values to encode. Table 3.1 shows the golomb code for m=5.

n	q	r	Codeword	n	q	r	Codeword
0	0	0	000	8	1	3	10110
1	0	1	001	9	1	4	10111
2	0	2	010	10	2	0	11000
3	0	3	0110	11	2	1	11001
4	0	4	0111	12	2	2	11010
5	1	0	1000	13	2	3	110110
6	1	1	1001	14	2	4	110111
7	1	2	1010	15	3	0	111000

Table 3.1: Golomb code for m=5

To encode an integer n using the Golomb code with parameter m, we first compute $\lfloor \frac{n}{m} \rfloor$ and output this using a unary code. Then we compute the remainder n mod m and output that value using a binary code which is adjusted so that we sometimes use $\lfloor log_2 m \rfloor$ bits and sometimes $\lceil log_2 m \rceil$ bits.

3.7.5 Rice Codes

Rice coding is the same as Golomb coding except that only a subset of parameters can be used, namely the powers of 2. In other words, a Rice code with the parameter k is equal to Golomb code with parameter $m = 2^k$. Because of this the Rice codes are much more efficient to implement on a computer. Division becomes a shift operation and modulo becomes a bit mask operation.

3.8 Summary

In this chapter, we have introduced the subject of data compression. Compression algorithms can be lossless or lossy. Lossy algorithms achieve better compression but are computationally expensive.

Chapter 4

Floating-point data and Compressability Issues

Compression algorithms, whether it is lossy or lossless, reduce the number of bits required to encode a symbol. The amount of compression achieved depends on nature of redundancies in the data and how well the algorithm exploits it. The type of algorithm to use for a particular application also depends on the nature of the data in concern. An algorithm which works well for text may not achieve the same results with image data. All these algorithms take advantage of the redundancies inherent in the data. For example, statistical methods like Huffman coding assign shorter codewords to symbols with more probability thus reducing the average number of bits required [3]. Dictionary-based algorithms like LZW achieve compression by replacing frequently occuring patterns with their index in the dictionary [11]. Transform based algorithms like JPEG transform the image data to a form which is easily compressible [12].

But in case of floating-point data, it poses a unique challenge in terms of compressability because of their structure. Generally, a floating point number is represented by a *sign bit*, an *exponent* part and a *mantissa* part, and these are not byte aligned in memory. In this chapter, we will describe the ubiquitous IEEE754 specification of floating-point numbers and discuss compressability issues with regards to floating-point data.

4.1 IEEE754 based floating point Representation

IEEE Standard 754 floating point representation is the most common representaion for real numbers on computers [5]. It has three components: the *sign* bit, the *exponent* and the *mantissa*. The base of the exponent is 2. It is implicit and need not be stored. Depending on the total number of bits or bytes used in the representaion, we have *single-precision* floating point numbers and *double-precision* floating point numbers. Figure 4.1 shows the layout of single-precision and double-precision formats, respectively.

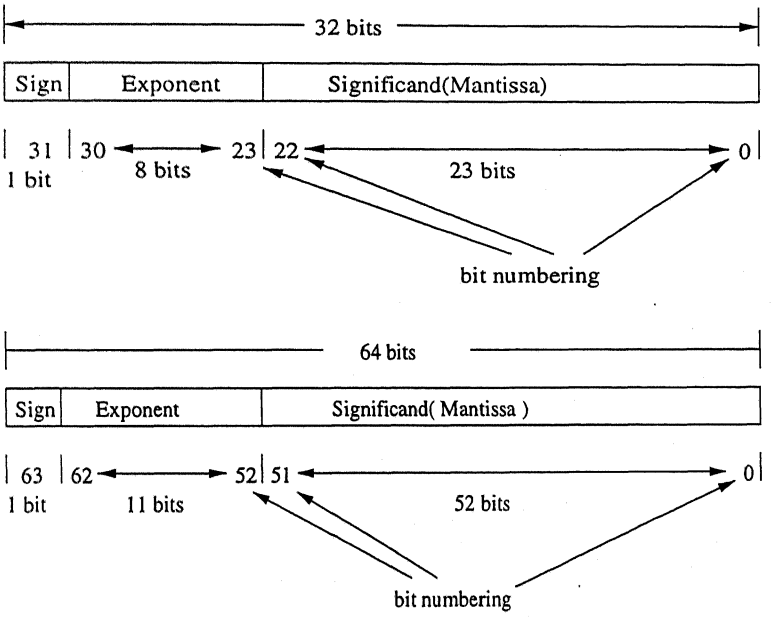


Figure 4.1: IEEE single-precision and double-precision floating-point number format

The sign bit indicates the sign of the number. A 0 implies positive number and 1 implies negative number. It is the first bit of the most significant byte.

The exponent field needs to represent both positive and negative values. This is done by use of a quantity called *bias*. The value stored in the bias is subtracted from the exponent field to get the actual exponent. For single-precision numbers, the bias is 127 and for double-precision numbers, the bias value is 1023. So, a zero exponent value is stored as 127 and 1023 for single-precision and double-precision numbers

respectively. If the exponent is negative, the value stored in the exponent field will be less than the bias value, and, if the exponent is positive, it will be greater than the bias value. For example, a value of 134 in the exponent field implies the actual exponent is 134-127 or 7 in single-precision.

The mantissa or significand field represents the precision bits of the number. It takes 23 bits in single-precision numbers and 52 bits in double-precision numbers as shown in Figure 4.1. A number can be represented in many ways in the scientific notation. For example, the number 23.1234 can be represented as any of the following:

$$23.123 \times 10^0$$
$$.23123 \times 10^2$$
$$2312.3 \times 10^{-2}$$
$$2.3123 \times 10^1$$

Because of this, floating-point numbers are represented in *normalized* form. In the normalized form, the radix point is placed after the first non-zero digit in the mantissa and the exponent is adjusted appropriately. For the above example, 2.3123×10^1 is in normalized form. The IEEE standard also stores in normalized form and it has an added advantage. In binary system, the only non-zero digit is 1. So, in normalized form, the 1 may be assumed to be present and need not be stored. This increases the precision of the mantissa by 1 bit.

Thus, a floating-point number can be formulated as:

$$(-1)^S * 2^{E-Bias} * 1.F$$

4.2 Review

Though there exist a large number of compression algorithms and tools based on them, very little work has been done on the topic of floating-point data compression. The reason could be that such data occur in very limited fields like space research. Text files are very common hence the special impetus on developing algorithms which compresses textual data. The Lempel-Ziv algorithms [13], [14] and their variants [11] were aimed for text compression. With the advent of world-wide-web, there was

felt the need algorithms to compress images, audio and video. In comparison, the need to develope algorithms for compressing floating-point data is very less.

Jericevic *et al* [7] have proposed a lossless compression algorithm for seismic data. The algorithm uses the idea of converting a signal data compression into a text compression problem. The conversion is based on *data slicing*, a reformatting process where the binary representation of a data set is rearranged, effectively separating compressible part from uncompressible parts. The reformatted data are then compressed using some known compression tool like the popular *gzip* program. They have reported a compression ratio in the range of 35-50%. The compression achieved is determined by the type of data. For floating-point data, the compression achieved is 36%.

In another work, Engelson *et al* [6] have proposed a lossless compression for floating-point data which is generated in scientific computing applications like simulations. They have used a variant of predictive coding called *delta-compression* which packs the high-order differences between adjacent data elements. Their algorithm assumes that the input data is *smooth*, i.e., there is substantial correlation between the adjacent data values. Let A be a sequence consisting of values a_i, $i = 1,...,n$, and let f:[1,n]$\rightarrow\mathcal{R}$, such that, $f(i)=a_i$. The sequence is called *smooth of order* m if every a_j $(j<m)$ can be well approximated by the extrapolating polynomial based on previous m values. In the simplest case, if the function f has very small and slow changes (is close to a polynomial of order zero, i.e., constant) then the sequence A can be very well compressed. In practice, such smoooth functions represent the solutions of ordinary differential equations and various continuous quantities that are computed in scientific simulation experiments.

A lossles method based on *wavelets* which enables progressive transmission of CFD (Computational Fluid Dynamics) data in PLOT3D format is presented in [10]. The floating-point data is first converted to double-precision floating-point format. It is then transformed using *Haar* wavelets and the transformed data is compressed using Huffman coding and transmitted progressively using spectral selection.

4.3 Problem with compression of Floating-Point Data

The set of real numbers $\mathcal{R}$ have the property of *infinite density*, i.e., there is a real number between any two real numbers. It has two parts: the *integral* part and the *frational* part. Let us assume that we have a subset of $\mathcal{R}$, say, $\mathcal{R}_1$, with values lying in the range $[P,Q]$. Let us represent the integral part of a real number by X_I and the fractional part by X_F. We can see that the X_Is for elements of $\mathcal{R}_1$ lie in the range $[P_I,Q_I]$ but the X_Fs have infinite number of possible values. As mentioned in Section 3.2.2, information theory deals with symbols sources that belong to a finite, predefined alphabet. So traditional compression schemes are not applicable to the alphabet $\mathcal{R}_1$.

Computers perform arithmetic in the binary system. Numeric data are also represented in binary. Real numbers are represented as floating point numbers. Numeric quantitites stored in binary can take on discreet values only and the range is determined by the number of bits used for the representation. For example, most computers use 4 bytes for representing *integers*, which implies that the maximum values that can be stored is 2^{32}(assuming unsigned integers). Since real number set is infinite, it is not possible for the computer to store all the possible values. The computer stores only an approximation using the IEEE754 standard for representing floating point numbers. Since compression algorithms deal with data stored on computers and the discreet nature of machine representation of numeric data imposes a limit on the infiniteness of real numbers, i.e., a single-precision floating point number is stored in 4 bytes on a computer. Hence, the traditional text and image compression algorithms can be applied to floating-point data. But the amount of compression achieved is not as much as achieved with text or image data. We tested the *gzip* program, Huffman coding and arithmetic coding on the input data set for our resampling routine. Since the input image sizes are very large(150MB per band), we read 1024x1024 pixels into a separate file and applied the algorithms. Following table shows the performance of each algorithm

Algorithm	Compressed size (in bytes)	Compression (in %)
Huffman Coding	3627189	86.48
Arithmetic Coding	3610220	86.07
Adaptative Huffman	3627357	86.48
gzip	2789905	66.51

The size of the input file is 4194304 bytes. The *gzip* program gives the best compression. There are two main reasons for the poor performance:

1. Due to the structure of the data. A floating point number consists of a *sign* bit, a biased *exponent* and a *mantissa*. These fields are not byte aligned in the memory.

2. Due to random nature of the fractional part of the number.

Random data cannot be compressed. Compression is possible only when there is some redundancy in the data. The idea of compression is to assign shorter codewords to more frequently occuring symbols and *vice-versa*. If there is randomness in the data, i.e, all symbols have equal probabilities, then compression is not possible. In the case of floating point numbers, randomness goes on increasing from the most significant bit to the least significant bit. The *data slicing* method proposed in [7] addresses this problem of randomness by dividing the 4 byte representation into a compressible part and an uncompressible part.

4.4 Summary

Traditional text and image compression schemes do not yield good compression when applied to floating point data. The poor compression due to the randomness of the lower fractional bits in the floating point representation and the structure of the data. The amount of compression is dependent on the nature of data.

Chapter 5

Our Approaches and Results

5.1 Introduction

As discussed in Chapter 4, floating point numbers are difficult to compress using the traditional compression algorithms. We have to devise algorithms which take into consideration the nature of the input data.

Although the input data to the resampling routine is in the form of floating point, it lies in the range [0.0,255.00]. A single-precision floating point number is represented by 4 bytes and consists of a *sign* bit, an 8-bit biased *exponent* and a 23-bit *significand* [5]. We can further divide the range of input pixel values into sub-ranges like [128.00,255.00], [64.00,127.00) and so on, such that the exponent value remains constant for a particular sub-range. Let x_i denote a pixel of the radiometrically corrected input image. Table 5.1 shows the sub-ranges of x_i and their corresponding exponent values. The sign bit is always 0 as all the values are positive. We can see that the exponent of x_i can take values in the range [127,134] for $1.00 \leq x_i \leq 255.00$. Generally, the exponent lies in that range. We have tried to take advantage of this and also the fact that there's a high degree of co-relation in image data, i.e., the arithmetic difference between any two neighbouring pixels tend to be very less, in our approaches.

Sub-Range	Exponent Value		
	Binary	Hex	Numeric
$128.00 \leq x_i \leq 255.00$	10000110	0x86	134
$64.00 \leq x_i < 128.00$	10000101	0x85	133
$32.00 \leq x_i < 64.00$	10000100	0x84	132
$16.00 \leq x_i < 32.00$	10000011	0x83	131
$8.00 \leq x_i < 16.00$	10000010	0x82	130
$4.00 \leq x_i < 8.00$	10000001	0x81	129
$2.00 \leq x_i < 4.00$	10000000	0x80	128
$1.00 \leq x_i < 2.00$	01111111	0x7f	127
$0.00 \leq x_i < 1.00$	01111111	0x7f	127

Table 5.1: Sub-ranges of x_i and its corresponding exponent value

In Section 2.1, we present an algorithm to encode a floating point pixel value in 3 bytes instead of 4 bytes thus reducing the input image size by 25%. This reduces the data transfer time. The algorithm is slightly lossy and produces some errors in the output. But the number of errors is minimal.

In Section 2.2, we present an algorithm to losslessly encode the exponent value. The algorithm assumes that there is a high degree of co-relation in the input image. The performance of this algorithm is slightly poorer than the lossy algorithm of Section 2.1.

In the distributed resampling model, input data is read by the Raddata process and the processing is done by the ResampProc processes. The data from the Raddata to the ResampProc process gets sent through the PVM daemon. In Section 2.3, we will describe a new routine which eliminates the data transfer overhead when the Raddata and ResampProc process are running on the same host.

We have tested our algorithms on a set of radiometrically corrected LISS-III 4 band scene data. The data is of type single-precision floating point. The first 3 bands are of size 149MB each and the 4th band is of size 17MB. Our implementation environment consists of 4 Intel PentiumIII PCs with 256MB RAM each, with Linux OS and connected through a 100Mbps switch.

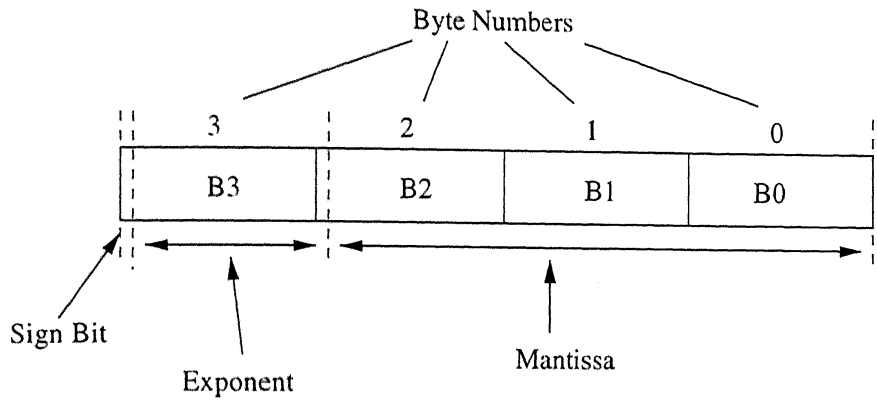


Figure 5.1: Byte-Wise Layout of Floating Point Number

5.2 Lossy Encoding of Exponent Byte

5.2.1 Introduction

Since, generally the exponent field's value lies in the range [127,134], which makes a total of 8 possibilities. We require 3-bits to encode these 8 possible values using fixed binary code as shown in Table 5.2.

Exponent	Binary code
134	000
133	001
132	010
131	011
130	100
129	101
128	110
127	111

Table 5.2: Fixed Binary Code for Exponent Field

Now the problem is with $0.00 \leq x_i < 1.00$, since the exponent value can take on any value in the range [0,127). We can add one more bit to the last codeword to indicate either the range $1.00 \leq x_i < 2.00$ or $0.00 \leq x_i < 1.00$. Say, if it is 0111, then $1.00 \leq x_i < 2.00$ and exponent value is 127, and, if it is 1111, then $0.00 \leq x_i < 1.00$. Table 5.3 shows the modified binary code. The exponent for $0.00 \leq x_i < 1.00$ can be stored explicitly.

Exponent	Binary code
134	000
133	001
132	010
131	011
130	100
129	101
128	110
127	1110
< 127	1111

Table 5.3: Modified Binary Code for Exponent Field

Using the binary code of Table 5.3, we can encode the exponent value using atmost 4 bits instead of 8 bits.

Let us look at the layout of IEEE 754 single-precision floating point number once again. Figure 5.1 shows the byte-wise layout. We will use the following notation throughout this chapter:

B_i: ith byte of the floating point number, i=0..3
B_i[j]: jth bit in the ith byte with the bits numbered from least significant to most significant and $j = 0..7$

The codewords for the exponents can be stored in the least significant bit positions of B_0. This way, we will lose some information. For $2.00 \leq x_i \leq 255.00$, the number of bits required is 3 bits and for $0.00 \leq x_i < 2.00$, the number of bits required is 4. So, we could lose atmost 4 bits of information. We can take advantage of the fact that the sign bit is always 0 and use $B_2[7]$ for storing 1 bit of the codeword. This way, the number of bits lost is reduced to atmost 3 bits.

The encoding process can be included in the radiometric correction phase, i.e., the routine which does the radiometric correction could output 3 bytes per pixel instead of the standard 4 bytes for floating point number. The Raddata process would read 3 bytes for each pixel thus saving I/O time. The ResampProc process upon receiving the data would decode it and generate the appropriate floating point number.

5.2.2 Algorithm

Table 5.3 shows the code we will be using to encode the exponents. The encoding process is very fast. For each floating point pixel x_i, we first obtain its exponent value. Say, x_i=65.443909, its byte-wise binary representation is

B_3	B_2	B_1	B_0
01000010	10000010	11100011	01001000

The exponent value is 133 or 10000101 in binary as we can see from Table 5.1. The codeword corresponding to this exponent value is 001 from Table 5.3. In fact, we don't need to maintain a code table. The value obtained by subtracting the exponent value from 134 gives the corresponding codeword. In the case of 133, 134-133 is equal to 1 or 00000001 in binary. The last 3 bits of the binary representation gives the codeword for the exponent, i.e., 001. We write this code onto the last three least significant bit positions of B_0. Before doing that, we copy $B_0[2]$ to $B_2[7]$, thus thus reducing the number of bits lost to 2 instead of 3. So, the final sequence of bytes becomes

B_2	B_1	B_0
00000010	11100011	01001001

In the case of $0.00 \leq x_i < 1.00$, we can copy a part of the exponent into B_0 and a part into B_1. We have observed that for small values of x_i, the error tolerance is more. So, copying the whole exponent field does not make much difference for $0.00 \leq x_i < 1.00$.

So, the pixels are written as the byte sequence $B2, B_1$ and B_0. These 3 bytes have enough information to recover the original floating point number with some loss of accuracy. The decoding process is just the reverse of encoding. We first obtain the last 3 bits of B_0. Subtracting from 134 we get the exponent value. For the above example, the last bits are 001, which means 1. Subtracting from 134, we get 133, which is the exponent of original floating point number 65.443909. We copy back $B_2[7]$ to $B_0[2]$ thus reducing the loss of information to 2 bits. Then, the bytes

are simply copied back and the exponent is adjusted so that it spans B_3 and B_2. The binary representation after decoding becomes

B_3	B_2	B_1	B_0
01000010	10000010	11100011	01001001

Only the last 2 bit positions of B_0 are lost as a result of the lossy encoding.

Encoding Algorithm

```
for each pixel x_i do
    Obtain B_3, B_2, B_1 and B_0 from x_i
    Exponent = GetExponentValue(B_3,B_2)
    code = 134 - Exponent
    if code < 7 then /* x_i:[1.00,255.00] and E:[128,134] */
       /* Copy the bit number 2 of B_0 to B_2 at bit position 7 */
       B_2[7] = B_0[2]
       /* Write last 3 bits of code to B_0 */
       WriteCode(B_0,code,3)
    else if index == 7 then /* x_i:[1.00,2.00) */
            /* Copy B_0[3] to B_2[7] */
            B_2[7] = B_0[3]
            /* Write the code 1111 to B_0 */
            WriteCode(B_0,0x07,4)
         else /* x_i:[0.00,1.00)
            /* Copy B_1[2] to B_2[7] */
            B_2[7] = B_1[2]
            /* Write the code 0111 to B_0 */
            WriteCode(B_0,0x0f,4)
            Copy Exponent[j], j=0..3 to B_0[k], k=4..7
            Copy Exponent[j], j=4..6 to B_1[k], k=0..2
         endif
```

```
        endif
    endfor
```

- *Decoding Algorithm*

```
Input: B2, B1, B0 Output: xi
    code = GetCode(B0)
    if code < 7 then /* xi:[1.00,255.00] and E:[128,134] */
        /* Copy the B2[7] to B0[2] */
        B0[2] = B2[7]
        /* Write the exponent value */
        Exponent = 134 − code
        WriteFloatNumber(Exponent,B2, B1, B0,xi)
    else if code == 7 then /* xi:[1.00,2.00) */
            /* Copy the B2[7] to B0[3] */
            B0[3] = B2[7]
            WriteFloatNumber(127,B2, B1, B0,xi)
        else /* xi:[0.00,1.00)
            Copy bits B0[j], j=4..7 to Exponent[k], k=0..3
            Copy bits B1[j], j=0..2 to Exponent[k], k=4..6
            WriteFloatNumber(Exponent,B2, B1, B0,xi)
        endif
    endif
```

- *Obtaining the Exponent Value*

The exponent field spans B_3 and B_2 as we can see in Figure 5.1. We can extract the exponent field's value by left-shifting B_3 by 1 bit and filling the least-significant bit position of B_3 by the most-significant bit of B_2. For example, let us consider the floating-point number 19.165432. Its byte-layout is given below

B_3	B_2	B_1	B_0
01000001	10011001	01010010	11001110

Left-shifting B_3 by 1 bit, we get

10000010

The most significant bit of B_2 is 1. Replacing the least-significant bit of B_3 by 1, we get

E=10000011

Following is the algorithm for the *GetExponentValue()* function:

Input: B_3, B_2
Begin
Left-shift B_3 by 1 bit position
$B_3[0] = B_2[7]$
End

The *WriteFloatNumber()* Function

Once we have the exponent value and the remaining bytes, the corresponding floating point number can be obtained by simply copying the bytes to their corresponding counterparts in the pixel. The exponent byte has to be right-shifted and its least significant bit has to copied to the $B_3[7]$.

Input: E,$A_2, A_1 and A_0$ **Output:** the floating-point pixel value, x_i
Begin
Obtain B_3, B_2, B_1, B_0 for pixel x_i
$B_2 = A_2$ $B_1 = A_1$ $B_0 = A_0$
B_3 = E
$B_2[7] = B_3[0]$
Right-shift B_3 by 1 bit position

End

5.2.3 Experimental Results

Figure 5.2 shows performance of our routine compared to the original routine with N number of nodes in the PVM configuration, N varying from 1 to 4.

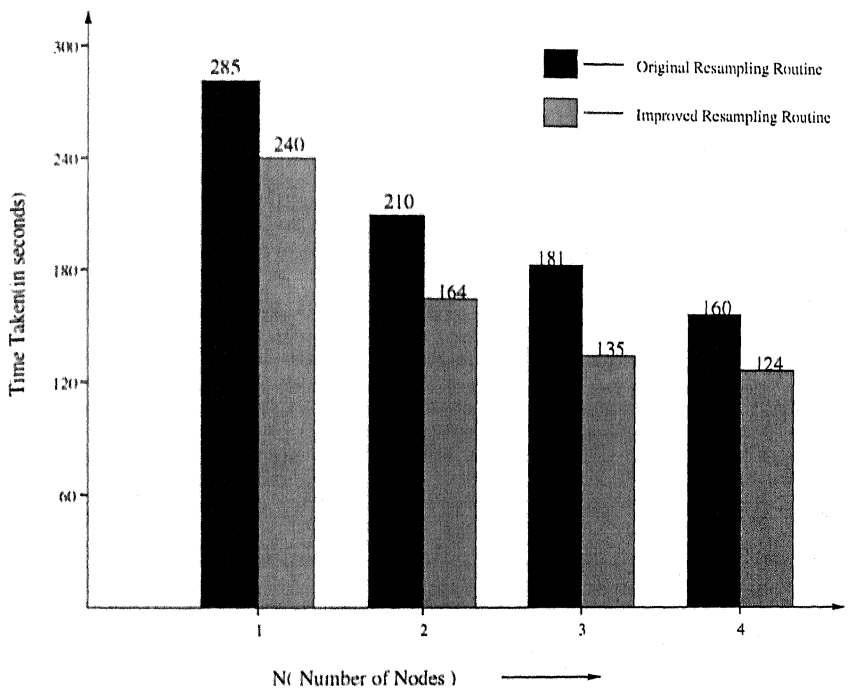


Figure 5.2: Performance Graph with Lossy Encoding

Error

Due to the lossy encoding, some error is introduced in the output. The error with our algorithm is atmost 1, i.e., the output pixel values of our algorithm differs from the original output pixels by atmost 1. For example, say, the original value is 123, then our algorithm gives a pixel value of 122.

The total number of errors in the output with our algorithm is

Band No.	No. of Errors
Band 1	77
Band 2	44
Band 3	65
Band 4	48

The total number of output pixels per band is 35536896.

5.3 Lossless Online Encoding of Exponent Byte

Generally, pixels in an image have a very high degree of co-relation, i.e, neighbouring pixels do not differ much. We have observed that this property holds true for the input data to the resampling routine too. There are long sequences of constant exponent values in the input image. Taking advantage of this property, we have developed an algorithm which is a variant of the RLE algorithm. It differs from the RLE algorithm in that does not encode the length of a sequence of repeating values. In RLE algorithm, a sequence of same symbols is encoded using two bytes. The first byte is the length of the sequence and the 2nd byte is the symbol which is repeated in the sequence (refer to Section 3.5). The input stream may not always contain such repeating sequences, so some mechanish is required to distinguish between RLE encoded sequence and symbols which have not been encoded. That is achieved by the use of a special control symbol. So, it requires a total of 3 bytes: 1 control character, 1 byte representing the length of the sequence and 1 byte representing the symbol itself. In our algorithm, we have eliminated the need for the control character and the length-byte.

5.3.1 Algorithm

In the Raddata process, the input data is read the following manner

```
for each band do
  for each scanline do
    READ N no. of pixels into buffer
  endfor
```

endfor

In our algorithm, the encoding of the exponent value is done using the same loop structure. For each scanline of data read, the exponent value of the first pixel, x_1 is obtained and stored in an array called EXP[ ]. The bytes B_2, B_1 and B_0 are stored, in that order, in another array called BUFFER[ ]. Two pointers are maintained for the two arrays, namely *exp_ptr* and *buffer_ptr*. The pointers are initialised to zero. The exponent value of x_1 is stored in a variable called E_{old}. For the remaining pixels in the scanline, i.e., x_i, i =2 to N, the exponent value of the ith pixel, E_{new}, is compared with E_{old}. If they are equal, a 0 is copied into $B_2[7]$. If the values differ, a 1 is copied into $B_2[7]$ and E_{new} is copied to E_{old}. Then, the new exponent value is copied to EXP[ ] and the bytes B_2, B_1 and B_0 are copied to BUFFER[ ]. The process is repeated for each scanline for each band. The arrays BUFFER[ ] and EXP[ ] are sent to the ResampProc process along with other information about the number of bands, number of scanlines and number of pixels, N, per scanline.

The code for the decoding process has to be included in the ResampProc process. It is just the reverse of the encoding process. It has the same loop structure as the encoding process. The pointers *exp_ptr* and *buffer_ptr* are initialised to zero. EXP[0] is the exponent value of the first pixel of the first scanline of the first band, and, BUFFER[0], BUFFER[1] and BUFFER[2] are x_1's B_2, B_1 and B_0, respectively. So, x_1, is easily obtained. Then, for the remaining pixels in a scanline, i.e., x_i, i=2 to N, we check whether $B_2[7]$ is set or not. If it is not set, then we simply obtain B_2, B_1 and B_0 from BUFFER[ ] and use WriteFloatNumber() to obtain the pixel value. If $B_2[7]$ is set, then we increment *exp_ptr* and EXP[*exp_ptr*] is the new exponent value.

The algorithm assumes that in a scanline there is high degree of co-relation among the pixels. The worst case occurs when no two consecutive pixels in a scanline have the same exponent value. In such a case, the size of the EXP[ ] array is same as N, the number of pixels in the scanline. Suppose, *NO_BANDS* is the number of bands, *NO_SCANS* is the number of scanlines per band and N is the number of pixels per scanline. Then, the total size of buffer required is

NO_BANDS*NO_SCANS*(4*N)

In case of our algorithm, the total size of buffer required is the sum of the sizes of BUFFER[ ] and EXP[ ]. Size of BUFFER[ ] is 3*N and the worst-case size of EXP[] is equal to N. Thus, in the worst case scenario, the size of buffer for our algorithm is

NO_BANDS*NO_SCANS*(3*N+N) = NO_BANDS*NO_SCANS*(4*N)

which is same as the original routine's.

- *Encoding*

Create two arrays: **BUFFER[ ]** and **EXP[ ]**
buffer_ptr = 0, exp_ptr = 0
for each band **do**
  for each scanline **do**
    Obtain $B_3, B_2, B_1 and B_0$ for x_1, the first pixel in the scanline
    E_{old} = GetExponentValue(B_3,B_2)
    EXP[exp_ptr] = E_{old}
    Increment exp_ptr by 1
    BUFFER[buffer_ptr] = B_2
    BUFFER[buffer_ptr+1] = B_1
    BUFFER[buffer_ptr+2] = B_0
    Increment buffer_ptr by 3
    for each pixel, x_i, $i = 2$ to N **do**
      Obtain B_3, B_2, B_1, B_0 for pixel x_i
      E_{new} = GetExponentValue(B_3,B_2)
      if E_{new} is equal to E_{old} **then** $B_2[7] = 0$
      else
        $B_2[7] = 0$
        $E_{old} = E_{new}$
      endif
      EXP[exp_ptr] = E_{old}
      Increment exp_ptr by 1
      BUFFER[buffer_ptr] = B_2

```
            BUFFER[buffer_ptr+1] = B1
            BUFFER[buffer_ptr+2] = B0
            Increment buffer_ptr by 3
        endfor
    endfor
    Increment exp_ptr by 1
endfor
```

- *Decoding*

```
buffer_ptr = 0, exp_ptr = 0
for each band do
    for each scanline do
        Exponent = EXP[exp_ptr]
        B2 = BUFFER[buffer_ptr]
        B1 = BUFFER[buffer_ptr+1]
        B0 = BUFFER[buffer_ptr+2]
        Increment buffer_ptr by 3
        WriteFloatNumber(Exponent,B2,B1,B0,x1)
        for i = 2 to N do
            B2 = BUFFER[buffer_ptr]
            if B2[7] is equal to 1 then
                Increment exp_ptr by 1
                Exponent = EXP[exp_ptr]
            endif
            B1 = BUFFER[buffer_ptr+1]
            B0 = BUFFER[buffer_ptr+2]
            Increment buffer_ptr by 3
            WriteFloatNumber(E,B2,B1,B0,xi)
        endfor
    endfor
```

Increment exp_ptr by 1

endfor

5.3.2 Experimental Results

Figure 5.3 shows the performance of this algorithm. The number of nodes vary from 2 to 4.

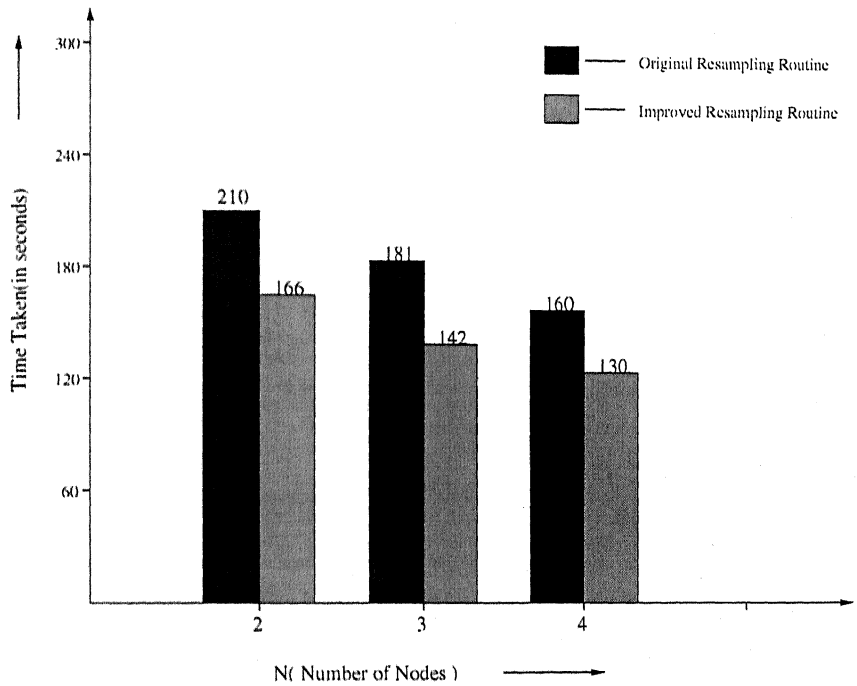


Figure 5.3: Performance Graph with Lossless Encoding

The algorithm has a slightly poorer performance compared to the lossy algorithm. That is because of the overhead of online encoding the exponent values. In the lossy algorithm, the exponent value is already encoded, so there's only the overhead of decoding. While in this algorithm, there is the overhead of both encoding and decoding.

5.4 Eliminating Master-to-Master Communication Overhead

In the distributed resampling model, the *Raddata* routine reads the data from the input files and sends it to the *ResampProc* routine. The ResampProc routine performs the correction and sends the output to the *Receiver* process which writes it to the output files. The same procedure is followed even if the Raddata routine and the ResampProc routine are executing on the same host, i.e, the master host. Although the time taken for sending the data to the ResampProc process on the same host is lesser than the time taken to send data to a ResampProc process on a different host, it has a finite overhead. This is because the data gets sent through the PVM daemon which uses a socket connection for sending the data. It doesn't care whether the process is running on the same host or different host.

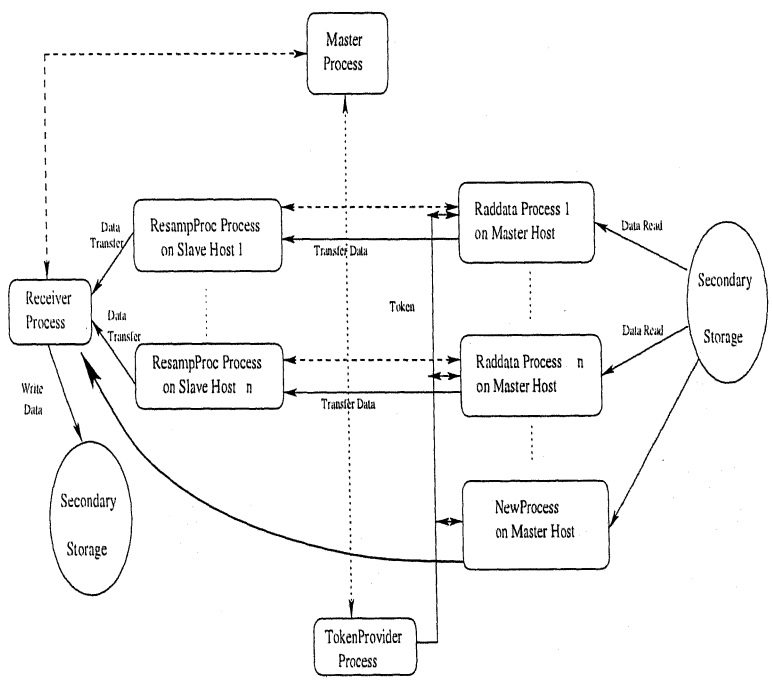


Figure 5.4: Resampling Model with *NewProcess*

We have developed a new routine which eliminates the master-to-master communication overhead. We will call the new routine *NewProcess*. This routine reads

the data from the input files, performs the corrections and sends the corrected data to the Receiver process. The master process checks if the next bunch is to be processed on the same host, if so, it spawns *NewProcess*. For different hosts, the earlier procedure is followed, i.e., Raddata's and ResampProc's are spawned. Figure 5.4 shows the process distribution with the inclusion of *NewProcess* in the distributed resampling model.

We used the *NewProcess* together with the algorithms described in Section 2.1 and Section 2.2. In case of the lossy algorithm (Section 2.1), *NewProcess* saves just the data transfer time, but in case of the lossless algorithm (Section 2.2), *NewProcess* saves the encoding time too, as their is no need for compressing the image data if the processing has to be done on the master host. Figure 5.5 and Figure 5.6 shows the performances of the lossy and lossless algorithms, respectively, with the inclusion of *NewProcess*.

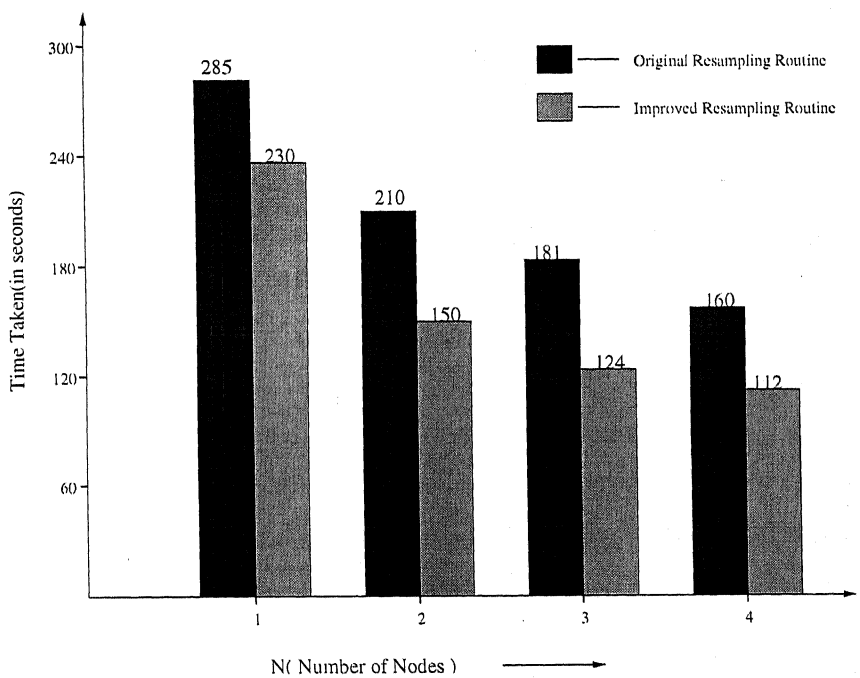


Figure 5.5: Performance Graph of Lossy Algorithm with *NewProcess*

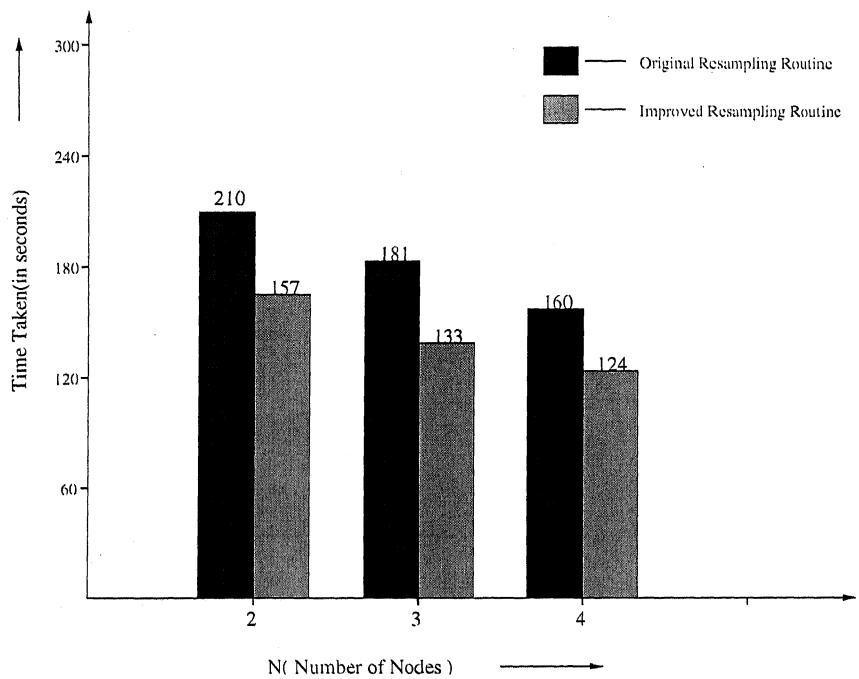


Figure 5.6: Performance Graph of Lossless algorithm with *NewProcess*

5.5 Summary

We have presented a lossy and a lossless algorithm for compressing the input image. The lossy algorithm shows better performance than the lossless one. It produces some errors in the output but it is negligible. The lossless algorithm can be used when no error is tolerable. In both the algorithms, the original floating point data is broken into bytes and those bytes are sent over the network. Thus, it eliminates the data incompatibility problem. We have also implemented a new routine, namely *newProcess*, which eliminates the master-to-master communication overhead.

Chapter 6

Conclusions and Future Work

The performance of any distributed computing application is affected by the follwing factors:

- Hardware related, that is, memory, processor speed and the channel capacity.
- Underlying TCP/IP overhead.
- PVM induced latency.
- Size of data and data transfer overhead.

In our work we have dealt with the last issue. We have presented a lossy as well as lossless algorithm which give better compression without taking much time. We have concentrated on the exponential field of the floating point number in our algorithms. There is scope for developing efficient algorithms which take into account the fact that there is usually some redundancy in the higher order bits of the *mantissa*.

PVM induces its own overhead and some overhead is also incurred due to the protocol(TCP/IP). The concept of *Active Messages* has been used to tackle the network related delays. There is scope for improvisation of the resampling routine in this field too.

In the distributed model, the *ResampProc* processes do the processing and send the output to the *Receiver* process. The output consists of values in the range [0..255]. Compression techniques could be applied to the output too.

Appendix A

A.1 Parallel Virtual Machine

Parallel Virtual Machine(PVM) is a software system with the help of which a heterogeneous collection of computers connected over a network can be treated as a single parallel machine. PVM system has become very popular for heterogeneous network computing. Heterogeneous Network Computing means that the computers on the network may be of different architectures, support different data formats and may have different computational speeds. Also, processing load may vary from computer to computer and network load may vary with time. PVM presents an integrated picture of this heterogeneity.

To solve a large problem, we divide it into several smaller subproblems and using PVM, employ several computers over the network to solve those subproblems. A computer integrated into the PVM environment is called a *host*. PVM system consists of two parts :

- PVM Daemon : It is commonly referred to as *pvmd*. It resides on all computers that are integrated into the PVM environment. A computer over the network can be configured into the PVM environment by running a PVM daemon on it.
- Library : This is a set of interface routines provided by PVM system. The user writes application over PVM with the help of these routines. The library consists routines for message passing, spawning tasks, controlling and

coordinating tasks, dynamically adding and deleting hosts from the PVM environment etc.

The applications over PVM are written as a set cooperating *tasks*. These tasks access the resources of PVM through a library of standard interface routines [15]. Each task is assigned a *task id*(tid) by the pvmd which is unique across the entire PVM environment. Typically, the user has to start one task manually on one of the hosts over PVM. This task, in turn, spawns several tasks on different hosts. The tasks communicate with each other by sending and receiving messages using PVM routines provided for this purpose. The source and destination tasks for the messages are specified by their tid. Also, each message has a *tag* associated with it which differentiates between the messages sent from the same source to the same destination. There is no limitation on the number and the size of messages, imposed by PVM. The size of the message is limited by the amount of memory available on a host.

Some main features of PVM are as follows :

- Dynamic configuration of PVM environment : Hosts can be added and deleted during the course of execution of any task.
- Cooperation via message passing : Tasks cooperate and synchronize by explicitly passing messages to each other. User has to take care of cooperation among the tasks while writing the application.
- Heterogeneous support : PVM supports a wide variety of architectures. A complete list of architectures supported by PVM is given in [15].
- Interoperability : Tasks running on different architectures can communicate with each other. Also, applications written in different programming languages (PVM supports C, C++ and Fortran) can communicate with each other in PVM [17]
- Fault Tolerance : A task can register with PVM to be notified if the status of a particular host changes or a task fails.

Other details about PVM and the library of interface routines are given in [15].

A.2 PVM Resources

Some important sites on the web related to PVM are :

- http://www.netlib.org/pvm3/ : The source code and binaries for the latest version of PVM are available on this site. Also available here are FAQs, user manuals and example programs.
- http://www.epm.ornl.gov/pvm/ : News about various projects on PVM, latest developments and releases, relevant papers etc is available on this site. Also hosts articles from "comp.parallel.pvm" newsgroup.

[9] M. Goldberg. Digital Image Processing of Remotely Sensed Imagery. *Digital Image Processing*, J.C. Simon and R.M. Haralick ed:383-437, D. Reidal Publishing Company, 1981.

[10] Aaron Gregory Trott. *The Application of Wavelets to Lossless Compression and Progressive Transmission of Floating Point Data in 3-D Curvilinear Grids.* Master's Thesis, Department of Electrical and Computer Engineering, Mississippi State University, May 1996.

[11] T. A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, pages 8-19, June 1984.

[12] William B. Pennebaker. *JPEG Still Image Data Compression Standard.* Van Nostrand Reinhold, New York, 1992.

[13] J. Ziv and A. Lempel. A Universal Algorithm for Data Compression. *IEEE Transactions on Information Theory*, IT-23(3):337-343, May 1977.

[14] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, IT-24(5):530-536, September 1978.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press,1994.

[16] Ravi P. Gupta. *Remote Sensing Geology.* Springer-Verlag Berlin, Heidelberg, 1991.

[17] J. A. Kohl, G. A. Geist and P. M. Papadopoulous. PVM and MPI : A Comparision of Features. *Calculateurs Paralleles*, 8(2), 1996.

[18] "XDR : External Data Representation Standard", Requests for Comments:1014, Network Working Group, Sun Microsystems Inc., June 1987.

[19] Mark Nelson. *The Data Compression Book.* BPB Publications, New Delhi, 1996.

[20] S. W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, IT-12:399-401, July 1966.

[21] J. J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20:198-203, May 1976.

[22] J. J. Rissanen and G. G. Langdon. Arithmetic Coding. *IBM Journal of Research and Development*, 23(2):149-162, March 1979.

[23] John R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers, San Francisco, California, 2000, pages 20-22.